University of Ljubljana, Faculty of Computer and Information Science

Text preprocessing

Prof Dr Marko Robnik-Šikonja

Natural language processing, Edition 2022

Lecture outline

- text preprocessing and normalization
- regular expressions and finite automata
- context dependent grammars
- Chomsky hierarchy

Read Chapter 2 of
Daniel Jurafsky & James H. Martin. Speech and Language Processing, 3rd edition, 2021.

Some slides from this source

Basic text preprocessing for the classical NLP pipeline

- document → paragraphs → sentences → words
- words and sentences ← POS tagging
- sentences ← syntactical and grammatical analysis

Text preprocessing

LooL :-)

- text normalization: transformation into a standard (canonic) form or any useful form, e.g., from non-standard language to standard
- upper/lower casing
- rediacritisation (for Slovene)
- notation of acronyms
- standard form of dates, time, and numbers
- stress marks, quotation marks, punctuation,
- non-informative words
- spelling, e.g., US or GB
- emoticons, emoji, hashtags, web links
- editing and presentation markup, e.g., html tags
- spelling correction
- usually start with tokenization

Primary quotation marks
in European languages
„x“
«x»
”x”
(also »x»)
”x”
(also »x»)
„x“
«x»
(“x” for names)
«X»
„x“
«x»
‘x’
“x”
(‘x’ preferred
in the UK;
“x” preferred
in the US)
„x“
»x«
„x”
rarely:
«x»
«X»
« x »
(with spaces)
„x“
„x”
„x”
«X»
“x”
«X»
(but “x” is
recommended
by El País)
«x»
in Brazil:
“x”
„x“
„x“
«X»
“x”
more maps at
jakubmarian.com

Token, type, term

- A *token* is an instance of a sequence of characters in some particular document that are grouped together as a useful semantic unit for processing.
- A *type* is the class of all tokens containing the same character sequence.
- A *term* is a (perhaps normalized) type that is included in the system's dictionary. The set of index terms could be entirely distinct from the tokens, e.g., term can be semantic identifiers in a taxonomy, but in practice they are strongly related to the tokens in the document. However, they are usually derived from them by various normalization processes.
- *To sleep perchance to dream*,
- 5 tokens, 4 types (2 instances of *to*)
- if *to* is omitted from the index (as a stop word), then there will be only 3 terms: *sleep*, *perchance*, and *dream*

Is this this simple?

```
## tokenizing a piece of text
doc = "I wrote this sentence"
for i, w in enumerate(doc.split(" ")):
    print("Token " + str(i) + ": " + w)
```

```
Token 0: I
Token 1: wrote
Token 2: this
Token 3: sentence
```

Words, lemmas, word forms, stems

- I do uh main- mainly business data processing
  - Fragments, filled pauses
- Seuss's cat in the hat is different from other cats!
  - **Lemma**: same stem, part of speech, rough word sense
    - cat and cats = same lemma
  - **Wordform**: the full inflected surface form
    - cat and cats = different wordforms

Words

- Lexical analysis (tokenizer, word segmented), not just spaces
- 1,999.00€ 1.999,00€!
- Ravne na Koroškem
- Port-au-prince
- Lebensversicherungsgesellschaft
- Generalstaatsverordnetenversammlungen
- I'm rock 'n' roll
- Languages without spaces (e.g., Chineese)

- Rules, finite automata, statistical models, dictionaries (of proper names), lexicons, ML models

How many words?

N = number of tokens

V = vocabulary = set of types, **|*V*|** is size of vocabulary

Heaps Law = Herdan's Law: $|V| = kN^{\beta}$ where often .67 < β < .75

i.e., vocabulary size grows with > square root of the number of word tokens

	Tokens = N	**Types = |V|**
Switchboard phone conversations	2.4 million	20 thousand
Shakespeare	884,000	31 thousand
COCA	440 million	2 million
Google N-grams	1 trillion	13+ million

Corpora

- Words don't appear out of nowhere.
- A text is produced by a specific writer(s), at a specific time, in a specific variety of a specific language, for a specific function.

Corpora vary along dimension like

- **Language**: 7097 languages in the world
- **Variety**, like African American Language varieties.
  - AAL Twitter posts might include forms like "*iont" (I don't)*
- **Code switching**, e.g., Spanish/English, Hindi/English:

S/E: Por primera vez veo a @username actually being hateful! It was beautiful:)

[For the first time I get to see @username actually being hateful! it was beautiful:)]

H/E: dost tha or ra- hega ... dont wory ... but dherya rakhe

[“he was and will remain a friend ... don’t worry ... but have faith”]

- **Genre:** newswire, fiction, non-fiction, scientific articles, Wikipedia
- **Author demographics**: writer's age, gender, race, socioeconomic status, etc.

Corpus datasheets

- **Motivation**: Why was the corpus collected, by whom, and who funded it?
- **Situation**: In what situation was the text written?
- **Collection process**: If it is a subsample how was it sampled? Was there consent? Pre-processing?
- +**Annotation process, Language variety, speaker demographics**

- See, e.g., corpora on Clarin.si

Text Normalization

- Most NLP task need text normalization:
  1. Segmenting/tokenizing words in running text
  2. Normalizing word formats
  3. Segmenting sentences in running text

Simple Tokenization in UNIX

- (Inspired by Ken Church's UNIX for Poets.)
- Given a text file, output the word tokens and their frequencies
- Command tr (translate)

```
tr -sc 'A-Za-z' '\n' < shakes.txt
      | sort
      | uniq -c
```

Change all non-alpha to newlines

Sort in alphabetical order

Merge and count each type

```
1945 A
  72 AARON
  19 ABBESS
   5 ABBOT
 ... ...
```

```
25 Aaron
 6 Abate
 1 Abates
 5 Abbess
 6 Abbey
 3 Abbot
... …
```

Issues in Tokenization

- Can't just blindly remove punctuation:
  - m.p.h., Ph.D., AT&T, cap’n.
  - prices ($45.55) and dates (01/02/06); URLs; (http://www.stanford.edu), hashtags (#nlproc), email addresses (someone@cs.colorado.edu).
- Clitics: a part of a word that can't stand on its own
  - we're → we are, French j'ai, l'honneur
- Can "Multiword Expressions (MWE) be words?
  - New York, rock ’n’ roll

Issues in Tokenization

- Finland's capital → Finland Finlands Finland's *?*
- what're, I'm, isn't → What are, I am, is not
- Hewlett-Packard → Hewlett Packard ?
- state-of-the-art → state of the art ?
- Lowercase → lower-case lowercase lower case ?
- San Francisco → one token or two?
- m.p.h., PhD. → ??

Tokenization in NLTK

Bird et al. (2009)

```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)    # set flag to allow verbose regexps
...     ([A-Z]\.)+        # abbreviations, e.g. U.S.A.
...   | \w+(-\w+)*        # words with optional internal hyphens
...   | \$?\d+(\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
...   | \.\.\.            # ellipsis
...   | [][.,;"'?():-_`]  # these are separate tokens; includes ], [
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

Tokenization: language issues

- French
  - ***L'ensemble*** $\rightarrow$ one token or two?
    - ***L*** ? ***L'*** ? ***Le*** ?
    - Want ***l'ensemble*** to match with ***un ensemble***

- German noun compounds are not segmented
  - ***Lebensversicherungsgesellschaftsangestellter***
  - 'life insurance company employee'
  - German information retrieval needs **compound splitter**

Word Tokenization in Chinese

- Also called **Word Segmentation**
- Chinese words are composed of characters called **hanzi**
- Each one represents a meaning unit called a morpheme.
  - Characters are generally 1 syllable and 1 morpheme.
  - Average word is 2.4 characters long.
- But deciding what counts as a word is complex and not agreed upon
- Standard baseline segmentation algorithm:
  - Maximum Matching (also called Greedy)
- So in Chinese it's common not to do word segmentation at all
- But in Thai and Japanese, it's required
- The standard algorithms are neural sequence models trained by supervised machine learning.

How to do word tokenization in Chinese?

•姚明进入总决赛 “Yao Ming reaches the finals”

•3 words?
•姚明　　进入　总决赛
•YaoMing reaches finals

•5 words?
•姚　　明　　进入　　总　　决赛
•Yao　Ming reaches overall　finals

•7 characters? (don't use words at all):
•姚　明　　进　　入　　总　　决　　赛
•Yao Ming enter enter overall decision game

Maximum Matching Word Segmentation Algorithm

- Given a wordlist of Chinese, and a string.

1) Start a pointer at the beginning of the string
2) Find the longest word in dictionary that matches the string starting at pointer
3) Move the pointer over the word in string
4) Go to 2

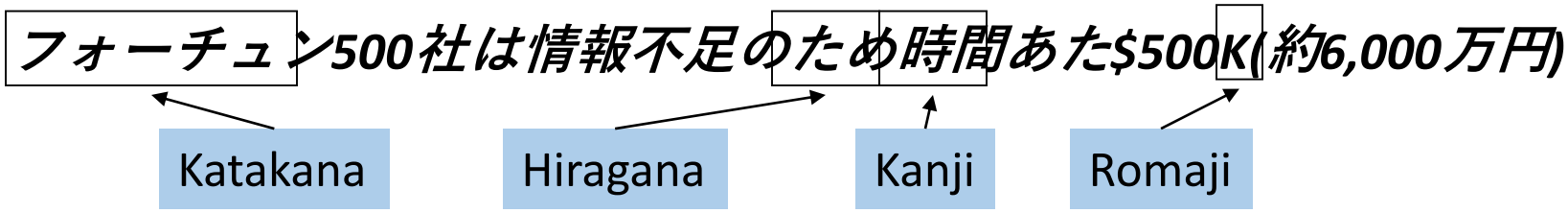
Max-match segmentation illustration

- Thecatinthehat — the cat in the hat
- Thetabledownthere — the table down there; theta bled own there
- Doesn't generally work in English!

- But works astonishingly well in Chinese
  - 莎拉波娃现在居住在美国东南部的佛罗里达。
  - 莎拉波娃 现在 居住 在 美国 东南部 的 佛罗里达
- Modern probabilistic segmentation algorithms even better

Tokenization: language issues

- Chinese and Japanese no spaces between words:
  - **莎拉波娃**现在居住在美国东南部的佛罗里达。
  - **莎拉波娃** 现在 **居住** **在** **美国** 东南部 **的** **佛**罗里达
  - Sharapova now lives in US southeastern Florida
- Further complicated in Japanese, with multiple alphabets intermingled
  - Dates/amounts in multiple formats



End-user can express query entirely in hiragana!

Subword Encoding tokenization

- Learn tokenization based on statistics
- Relevant for modern neural networks
- Use the data to tell us how to tokenize.
- **Subword tokenization** (because tokens are often parts of words)
- Can include common morphemes like *-est* or *-er*.
  - (A morpheme is the smallest meaning-bearing unit of a language; *unlikeliest* has morphemes *un-*, *likely*, and *-est*.)
- Essential for morphologically-rich languages such as Slovene

Subword tokenization

- Three common algorithms:
  - **Byte-Pair Encoding (BPE)** (Sennrich et al., 2016)
  - **unigram language modeling tokenization** (Kudo, 2018)
  - **WordPiece** (Schuster and Nakajima, 2012)
- All have 2 parts:
  - A token **learner** that takes a raw training corpus and induces a vocabulary (a set of tokens).
  - A token **segmenter** that takes a raw test sentence and tokenizes it according to that vocabulary

Byte Pair Encoding (BPE)

Let vocabulary be the set of all individual characters

= {A, B, C, D,…,a, b, c, d….}

- Repeat:
  - choose the two symbols that are most frequently adjacent in training corpus (say ‘A’, ‘B’),
  - adds a new merged symbol ‘AB’ to the vocabulary
  - replace every adjacent ’A’ ’B’ in corpus with ‘AB’.
- Until *k* merges have been done.

BPE token learner algorithm

function BYTE-PAIR ENCODING(strings C, number of merges k) **returns** vocab V

$V \leftarrow$ all unique characters in C # initial set of tokens is characters
for $i = 1$ **to** k **do** # merge tokens til k times
  $t_L, t_R \leftarrow$ Most frequent pair of adjacent tokens in C
  $t_{NEW} \leftarrow t_L + t_R$ # make new token by concatenating
  $V \leftarrow V + t_{NEW}$ # update the vocabulary
  Replace each occurrence of t_L, t_R in C with t_{NEW} # and update the corpus
return V

Byte Pair Encoding (BPE)

- Most subword algorithms are run inside white-space separated tokens.
- So first add a special end-of-word symbol '__' before whitespace in training corpus
- Next, separate into letters.

BPE token learner

Original (very fascinating 😍) corpus:

low low low low low lowest lowest newer newer newer newer newer newer wider wider wider new new

Add end-of-word tokens and segment:

corpus		vocabulary
5	l o w _	_, d, e, i, l, n, o, r, s, t, w
2	l o w e s t _	
6	n e w e r _	
3	w i d e r _	
2	n e w _	

BPE token learner

corpus	vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w
2 l o w e s t _	
6 n e w e r _	
3 w i d e r _	
2 n e w _	

Merge e r to er

corpus	vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er
2 l o w e s t _	
6 n e w er _	
3 w i d er _	
2 n e w _	

BPE

corpus	vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er
2 l o w e s t _	
6 n e w er _	
3 w i d er _	
2 n e w _	

Merge er _ to er_

corpus	vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er, er_
2 l o w e s t _	
6 n e w er_	
3 w i d er_	
2 n e w _	

BPE

corpus	vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er, er_
2 l o w e s t _	
6 n e w er_	
3 w i d er_	
2 n e w _	

Merge n e to ne

corpus	vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne
2 l o w e s t _	
6 ne w er_	
3 w i d er_	
2 ne w _	

BPE

The next merges are:

Merge	**Current Vocabulary**
(ne, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new
(l, o)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo
(lo, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low
(new, er_)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_
(low, _)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_, low_

BPE token learner algorithm

- On the test data, run each merge learned from the training data:
  - Greedily
  - In the order we learned them
  - (test frequencies don't play a role)
- So: merge every e r to er, then merge er _ to er_, etc.
- Result:
  - Test set "n e w e r _" would be tokenized as a full word
  - Test set "l o w e r _" would be two tokens: "low er_"

Normalization

- Need to “normalize” terms
  - Information Retrieval: indexed text & query terms must have the same form.
    - We want to match ***U.S.A.*** and ***USA***
    - uhhuh or uh-huh
    - Fed or fed
    - am, is be, are
- We implicitly define equivalence classes of terms
  - e.g., deleting periods in a term
- Alternative: asymmetric expansion:
  - Enter: ***window*** Search: ***window, windows***
  - Enter: ***windows*** Search: ***Windows, windows, window***
  - Enter: ***Windows*** Search: ***Windows***
- Potentially more powerful, but less efficient

Case folding

- Applications like IR: reduce all letters to lower case
  - Since users tend to use lower case
  - Possible exception: upper case in mid-sentence?
    - e.g., ***General Motors***
    - ***Fed*** vs. ***fed***
    - ***SAIL*** vs. ***sail***
- For sentiment analysis, MT, information extraction
  - Case is helpful (***US*** versus ***us*** is important)

Lemmatization

- Reduce inflections or variant forms to base form
  - *am, are, is* → *be*
  - *car, cars, car's, cars'* → *car*
- *the boy's cars are different colors* → *the boy car be different color*
- Lemmatization: have to find correct dictionary headword form
- Machine translation
  - Slovene hočem (‘I want’), hočeš (‘you want’) the same lemma as hoteti ‘want’

Morphology

- **Morphemes**:
  - The small meaningful units that make up words
  - **Stems**: The core meaning-bearing units
  - **Affixes**: Bits and pieces that adhere to stems
    - Often with grammatical functions
- Morphological Parsers:
  - Parse *cats* into two morphemes *cat* and *s*
  - Parse Spanish *amaren* ('if in the future they would love') into morpheme *amar* 'to love', and the morphological features *3PL* and *future subjunctive*.

Lemmatization

- Lemmatization is the process of grouping together the different inflected forms of a word so they can be analyzed as a single item.
- Lemmatization difficulty is language dependent i.e., depends on morphology
- *English*
  - *walk, walked, walking, walks, ne pa walker*
  - *go, goes, going, gone, went*
- *Slovene*
  - *priti, pridem, prideš, pride, prideva, prideta, pridejo, pridemo, pridete, pridejo, ne pa prihod, prihodnost, prihajanje, prišlec*
  - *vlak, vlaka, vlaku, vlakom, vlakov,vlakoma,vlakih,vlaki, vlake*
  - *jaz, mene, meni, mano*
  - *Gori na gori gori!*
  - *Gori, na gori gori!*
- Use rules, dictionaries, lexicons, machine learning models
- Ambiguity resolution may be difficult

  Meni je vzel z mize (zapestnico).
- Quick solutions and heuristics, in English just remove suffixes: *–ing, -ation, -ed, …*
- essential approach for morphologically rich languages (Slavic, Arabic, Turkish, Spanish, etc)

Dealing with complex morphology is sometimes necessary

- Some languages requires complex morpheme segmentation
  - Turkish
  - Uygarlastiramadiklarimizdanmissinizcasina
  - `(behaving) as if you are among those whom we could not civilize'
  - Uygar `civilized' + las `become'
    + tir `cause' + ama `not able'
    + dik `past' + lar 'plural'
    + imiz 'p1pl' + dan 'abl'
    + mis 'past' + siniz '2pl' + casina 'as if'

Stemming

- stem: the root or main part of a word, to which inflections or formative elements are added
- in English
- simple solution: remove affixes

for example compressed and compression are both accepted as equivalent to compress.

for exampl compress and compress ar both accept as equival to compress

- Stemmer operates on a single word *without* knowledge of the context, and therefore cannot discriminate between words which have different meanings depending on part of speech (meeting: a lemma is to meet or a meeting). Speed!
- Potter algorithm

Porter's algorithm

- Commonest algorithm for stemming English
  - Results suggest it's at least as good as other stemming options
- Conventions + 5 phases of reductions
  - phases applied sequentially
  - each phase consists of a set of commands
  - sample convention: *Of the rules in a compound command, select the one that applies to the longest suffix.*

Typical rules in Porter algorithm

- *sses* $\rightarrow$ *ss*
- *ies* $\rightarrow$ *i*
- *ational* $\rightarrow$ *ate*
- *tional* $\rightarrow$ *tion*

- Weight of word sensitive rules
- *(m>1) EMENT* →
  - *replacement* → *replac*
  - *cement* → *cement*

Porter's algorithm
The most common English stemmer

Step 1a

sses	→	ss	caresses	→	caress
ies	→	i	ponies	→	poni
ss	→	ss	caress	→	caress
s	→	ø	cats	→	cat

Step 1b

(*v*)ing	→	ø	walking	→	walk
			sing	→	sing
(*v*)ed	→	ø	plastered	→	plaster
…					

Step 2 (for long stems)

ational	→	ate	relational	→	relate
izer	→	ize	digitizer	→	digitize
ator	→	ate	operator	→	operate
…					

Step 3 (for longer stems)

al	→	ø	revival	→	reviv
able	→	ø	adjustable	→	adjust
ate	→	ø	activate	→	activ
…					

Errors of Porter algorithm

Errors of Commission		Errors of Omission	
organization	organ	European	Europe
doing	doe	analysis	analyzes
numerical	numerous	noise	noisy
policy	police	sparse	sparsity

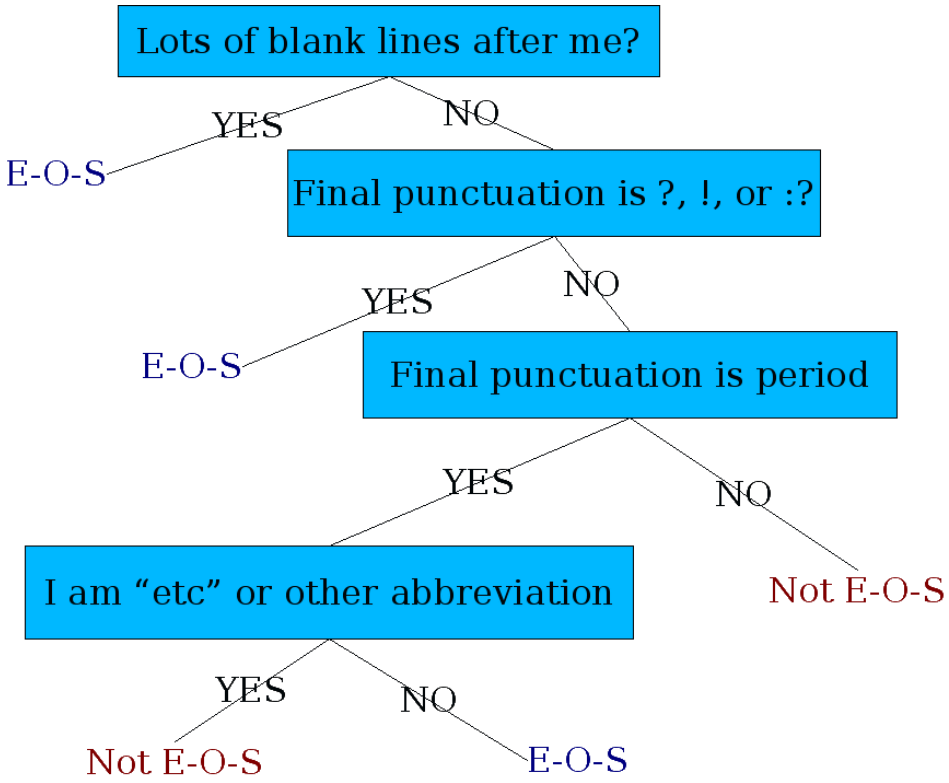
Sentences

- sentence delimiters – punctuation marks and capitalization are insufficient
- E.g., remains of 1. Timbuktu from 5c BC, were discovered by dr. Barth.
- Regular expressions, rules, manually segmented corpora

Sentence Segmentation

- !, ? are relatively unambiguous
- Period "." is quite ambiguous
  - Sentence boundary
  - Abbreviations like Inc. or Dr.
  - Numbers like .02% or 4.3
- Build a binary ML classifier
  - Looks at a "."
  - Decides EndOfSentence/NotEndOfSentence
  - Classifiers: hand-written rules, regular expressions, or machine-learning

Determining if a word is end-of-sentence: a Decision Tree



More sophisticated decision tree features

- Case of word with ".": Upper, Lower, Cap, Number
- Case of word after ".": Upper, Lower, Cap, Number

- Numeric features
  - Length of word with "."
  - Probability(word with "." occurs at end-of-s)
  - Probability(word after "." occurs at beginning-of-s)

Implementing Decision Trees

- A decision tree is just an if-then-else statement
- The interesting part is choosing the features
- Setting up the structure is often too hard to do by hand
  - Hand-building only possible for very simple features, domains
    - For numeric features, it's too hard to pick each threshold
  - Instead, structure usually learned by machine learning from a training corpus

Decision Trees and other classifiers

- We can think of the questions in a decision tree
- As features that could be exploited by any kind of classifier
  - Logistic regression
  - SVM
  - Neural Nets
  - etc.

Tools

- every NLP library has a tokenizer, sentence delimiter, lemmatizer, e.g., NLTK, spacy

- for Slovene
- https://www.cjvt.si/viri/
- https://github.com/clarinsi
- for nonstandard Slovene (twits, forum messages)
  - ***Nikola Ljubešić, Tomaž Erjavec, Darja Fišer:*** Orodja za procesiranje nestandardne slovenščine. V Fišer, D. (ur). 2018. Viri, orodja in metode za analizo spletne slovenščine. Ljubljana: Znanstveni založbi Filozofske fakultete Univerze v Ljubljani.

Regular expressions - a quick resume 1/3

- standard notation for characterizing text sequences
- used in all kinds of text processing and information extraction tasks
- many different syntaxes (Perl, grep, sed, awk, Python, etc)
- let‘s use regular expressions (RE) from python
- if A and B are REs then AB is RE
- a,b,…,z, A, B,… Z,0,1,…,9 are REs
- e.g. abeceda is RE
- . matches any character, e.g.: va.a matches vaba or vaza or vaya
- ^ matches the start of a string; ^.oga matches noga or joga, but not nadloga
- $ matches the end of a string
- * matches 0 or more repetitions of the previous RE: ab* matches a, ab, abb, …
- + matches 1 or more repetitions of the previous RE: ab+ matches ab, abb, … but not a

Regular expressions 2/3

- ? matches 0 or 1 repetitions of the previous RE: ab? matches a or ab
- *, + and ? are greedy: they match the longest possible string, e.g., <.*> on the string <a> b <c> matches the whole string
- *?, +?, ?? cause minimal matching of *, +, and ?, e.g.,.: <.*?> on the string <a> b <c> will match <a>
- {m} matches m repetitions of a previous RE: b{5} matches only bbbbb
- {m,n} matches from m to n repetitions of a previous RE
- {,n} is the same as {0,n}
- {m,} is the same as {m,∞}
- {m,n}? is a non-greedy variant of {m,n}
- \ is an escape character, it makes the next character special, e.g.,
  \\ matches \
  * matches *

Regular expressions 3/3

- [] represents a set of characters, e.g., [abc] matches a, b, or c; with [] we can represent a sequence of characters, e.g., [a-z] matches all lowercase letters from a to z
  special characters inside the set are not special, e.g., ?,+,*
- [^] (^ as the first character) represents a complement of a set, e.g., [^abc] matches all characters except a, b, and c
- | in A|B, where A and B are REs, means that RE matches A or B, several REs separated with | is tested from left to right, operator | is not greedy
- (…) matches RE in the parenthesis and marks a group, which can be used later or retrieved with \group_number
- (?aiLmsux), where after ? there are one or more letters means: a – only ASCII matches, i – ignore lower/uppercase, L – depend on the local settings, m – multi-line, s – the dot matches everything, etc. – check the manual
- many other useful details

Example

- Find me all instances of the word “the” in a text.

  `the`

  Misses capitalized examples

  `[tT]he`

  Incorrectly returns `other` or `theology`

  `[^a-zA-Z][tT]he[^a-zA-Z]`

Errors

- The process we just went through was based on fixing two kinds of errors:

1. Matching strings that we should not have matched (there, then, other)

False positives (Type I errors)

2. Not matching things that we should have matched (The)

False negatives (Type II errors)

Errors cont.

- In NLP we are always dealing with these kinds of errors.
- Reducing the error rate for an application often involves two antagonistic efforts:
  - Increasing accuracy or precision (minimizing false positives)
  - Increasing coverage or recall (minimizing false negatives).

Substitutions

- Substitution in Python and UNIX commands:

- `s/regexp1/pattern/`
- e.g.:
- `s/colour/color/`

Simple Application: ELIZA

- Early NLP system that imitated a Rogerian psychotherapist (Weizenbaum, 1966).

- Uses pattern matching to match, e.g.,:
  - `"I need X"`

  and translates them into, e.g.
  - `"What would it mean to you if you got X?`

Simple Application: ELIZA

Men are all alike.
IN WHAT WAY

They're always bugging us about something or other.
CAN YOU THINK OF A SPECIFIC EXAMPLE

Well, my boyfriend made me come here.
YOUR BOYFRIEND MADE YOU COME HERE

He says I'm depressed much of the time.
I AM SORRY TO HEAR YOU ARE DEPRESSED

How ELIZA works

- s/.* I'M (depressed|sad) .*/I AM SORRY TO HEAR YOU ARE \1/
- s/.* I AM (depressed|sad) .*/WHY DO YOU THINK YOU ARE \1/
- s/.* all .*/IN WHAT WAY?/
- s/.* always .*/CAN YOU THINK OF A SPECIFIC EXAMPLE?/

Summary

- Regular expressions play a surprisingly large role
  - Sophisticated sequences of regular expressions are often the first model for any text processing text
- For hard tasks, we use machine learning classifiers
  - But regular expressions are still used for pre-processing, or as features in the classifiers
  - Can be very useful in capturing generalizations

RE exercises

Write regular expressions for the following languages

- the set of all alphabetic strings;
- the set of all lower case alphabetic strings ending in a b
- the set of all strings with two consecutive repeated words (e.g., “Humbert
- Humbert” and “the the” but not “the bug” or “the big bug”);
- the set of all strings from the alphabet a,b such that each a is immediately
- preceded by and immediately followed by a b;
- all strings that start at the beginning of the line with an integer and that end at the end of the line with a word;
- all strings that have both the word grotto and the word raven in them (but not, e.g., words like grottos that merely contain the word grotto);

Formal Languages and Models

- Language: a (possibly infinite) set of strings made up of symbols from a finite alphabet
- Model of a language: can *recognize* and *generate* **all** and **only** the strings from the language
  - Serves as a definition of the formal language
- Alphabet Σ is a finite set of symbols, e.g., Σ ={0,1} or Σ={a,b,c,d}.
- String is a sequence of symbols from alphabet
- ε is an empty set
- $\Sigma \cup \Sigma\Sigma$ is a set of all strings of length 1 or 2
- Σ^* is a set of all strings from alphabet
- imprecise notation, e.g., 0 is a symbol and 0 is a string, depending on the context

Merrill, W., 2021. Formal Language Theory Meets Modern NLP. *arXiv preprint arXiv:2102.10094.* About formal languages and their relation with neural networks.

Language

- Language is a subset of Σ* for an alphabet Σ.
- Example: language of 0 and 1, where there are no two consecutive 1s
- L = {ε, 0, 1, 00, 01, 10, 000, 001, 010, 100, 101, 0000, 0001, 0010, 0100, 0101, 1000, 1001, 1010, . . . }

Chomsky Hierarchy

- Regular language
  - Model: regular expressions, finite state automata
- Context free language
- Context sensitive language
- Unrestricted language
  - Model: Turning Machine

Regular Expressions and Languages

- A regular expression pattern can be mapped to a set of strings
- A regular expression pattern defines a language (in the formal sense) – the class of this type of languages is called a regular language

An example of non-regular language

$L_1 = \{0^n1^n \mid n \geq 1\}$

$L_1 = \{01, 0011, 000111, \ldots\}$

An example

L_2 = {w | w $\in$ {(,)}* with balanced brackets}.

E.g.: (), ()(), (()), (()()),…

Context Free Grammars (CFG)

- A *context-free grammar* is a notation for describing languages.
- It is more powerful than finite automata or RE's, but still cannot define all possible languages.
- Useful for nested structures, e.g., parentheses in programming languages.
- Basic idea is to use "variables" to stand for sets of strings (i.e., languages).
- These variables are defined recursively, in terms of one another.
- Recursive rules ("productions") involve only concatenation.
- Alternative rules for a variable allow union.

Example: CFG for $\{ 0^n1^n \mid n \geq 1\}$

- Productions:
  - S -> 01
  - S -> 0S1
- 01 is part of a language
- if w is in the language, so is 0w1

Syntax

- Syntax = rules describing how words can connect to each other

- *that and after year last*
- *I saw you yesterday*
- *colorless green ideas sleep furiously*

- the kind of implicit knowledge of your native language that you had mastered by the time you were 3 or 4 years old without explicit instruction
- not necessarily the type of rules you were later taught in school.

Syntax

- Why should you care?
  - Grammar checkers
  - Question answering
  - Information extraction
  - Machine translation

CFG Formalism

- *Terminals* = symbols of the alphabet of the language being defined.
- *Variables* = *nonterminals* = a finite set of other symbols, each of which represents a language.
- *Start symbol* = the variable whose language is the one being defined.
- A *production* has the form variable -> string of variables and terminals.
- Convention:
  - A, B, C,… are variables.
  - a, b, c,… are terminals.
  - …, X, Y, Z are either terminals or variables.
  - …, w, x, y, z are strings of terminals only.
  - α, β, γ,… are strings of terminals and/or variables.

Example: Formal CFG

- Here is a formal CFG for $\{ 0^n1^n \mid n \geq 1\}$.
- Terminals = {0, 1}.
- Variables = {S}.
- Start symbol = S.
- Productions =

  S -> 01

  S -> 0S1

Derivations – Intuition

- We *derive* strings in the language of a CFG by starting with the start symbol, and repeatedly replacing some variable A by the right side of one of its productions.
  - That is, the "productions for A" are those that have A on the left side of the ->.

Derivations – Formalism

- We say $\alpha A \beta \Rightarrow \alpha\gamma\beta$ if $A \to \gamma$ is a production.
- Example: S -> 01; S -> 0S1.
- S => 0S1 => 00S11 => 000111.

Iterated Derivation

- =>* means “zero or more derivation steps.”
- Basis: α =>* α for any string α.
- Induction: if α =>* β and β => γ, then α =>* γ.

Example: Iterated Derivation

- S -> 01; S -> 0S1.
- S => 0S1 => 00S11 => 000111.
- So S =>* S; S =>* 0S1; S =>* 00S11; S =>* 000111.

Language of a Grammar

- If G is a CFG, then L(G), the *language of G*, is {w | S =>* w}.
  - Note: w must be a terminal string, S is the start symbol.
- Example: G has productions S -> ϵ and S -> 0S1.
- L(G) = $\{0^n1^n \mid n \geq 0\}$.
- Note: ϵ is a legitimate right side.

Context-Free Languages

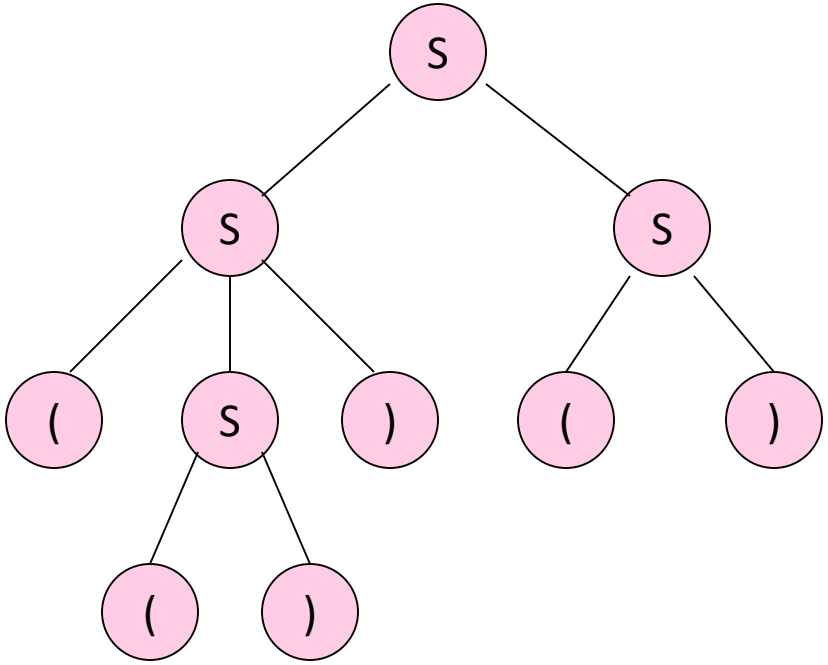
- A language that is defined by some CFG is called a *context-free language*.
- There are CFL's that are not regular languages, such as the example just given.
- But not all languages are CFL's.
- Intuitively: CFL's can count two things, not three.

Parse Trees

- *Parse trees* are trees labeled by symbols of a particular CFG.
- Leaves: labeled by a terminal or ϵ.
- Interior nodes: labeled by a variable.
  - Children are labeled by the right side of a production for the parent.
- Root: must be labeled by the start symbol.

Example: Parse Tree

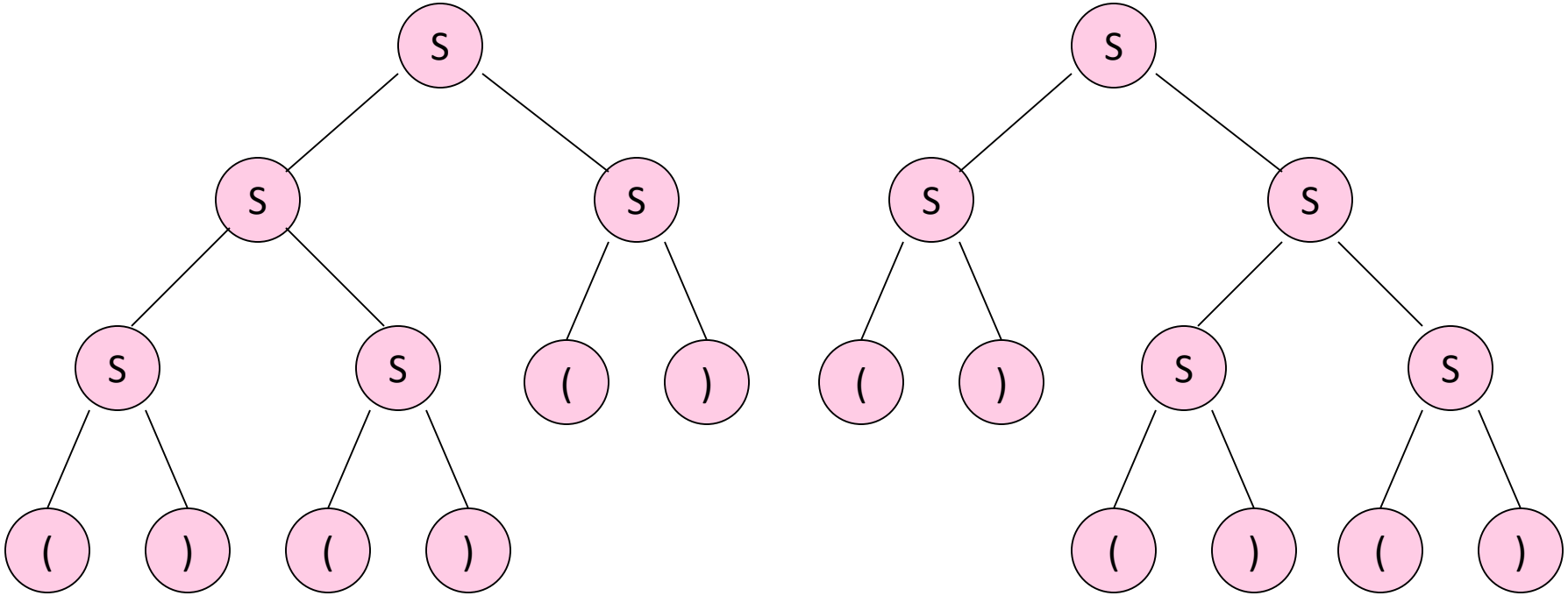
S -> SS | (S) | ()



Ambiguous Grammars

- A CFG is *ambiguous* if there is a string in the language that is the yield of two or more parse trees.
- Example: S -> SS | (S) | ()
- Two parse trees for ()()() on next slide.

Example



Ambiguity is a Property of Grammars, not Languages

- For the balanced-parentheses language, here is another CFG, which is unambiguous.

B -> (RB | ϵ

R ->) | (RR

B, the start symbol, derives balanced strings.

R generates strings that have one more right bracket than left.

Inherent Ambiguity

- It would be nice if for every ambiguous grammar, there were some way to "fix" the ambiguity, as we did for the balanced-parentheses grammar.
- Unfortunately, certain CFL's are *inherently ambiguous*, meaning that every grammar for the language is ambiguous.

Example: Inherent Ambiguity

- The language $\{0^i1^j2^k \mid i = j \text{ or } j = k\}$ is inherently ambiguous.
- Intuitively, at least some of the strings of the form $0^n1^n2^n$ must be generated by two different parse trees, one based on checking the 0's and 1's, the other based on checking the 1's and 2's.

One Possible Ambiguous Grammar

S -> AB | CD

A -> 0A1 | 01

B -> 2B | 2

C -> 0C | 0

D -> 1D2 | 12

A generates equal 0's and 1's

B generates any number of 2's

C generates any number of 0's

D generates equal 1's and 2's

And there are two derivations of every string with equal numbers of 0's, 1's, and 2's. E.g.:
S => AB => 01B =>012
S => CD => 0D => 012

Exercises

- Write CFG for a language

- $L(G) = \{\text{all words of a form } a^n b^m c^k, \text{where } n + m = k\}$

- $L(G) = \{\text{all words of a form } a^n b^m c^k, \text{where } n + k = m\}$

Chomsky Normal Form

- A CFG is said to be in *Chomsky Normal Form* if every production is of one of these two forms:
  1. A -> BC (right side is two variables).
  2. A -> a (right side is a single terminal).
- Theorem: If L is a CFL, then L – {Є} has a CFG in CNF.

Decision properties of CFG

1. $w \in L$
2. $L = \{\}$
3. L is infinite
4. $L_1 = L_2$
5. $L_1 \cap L_2 = \{\}$

Algorithm CYK – testing membership

- CYK: Cocke – Younger – Kasami
- CFG={V,T,S,P}
- answers the question $x \in L$ (or equivalently $S \Rightarrow^* x$)
- examples
  - is a given program correct according to the given grammar
  - is the given sentence grammatically correct
- requires CFG in Chomsky normal form
- $O(n^3)$, where $n = |w|$.

CYK Algorithm

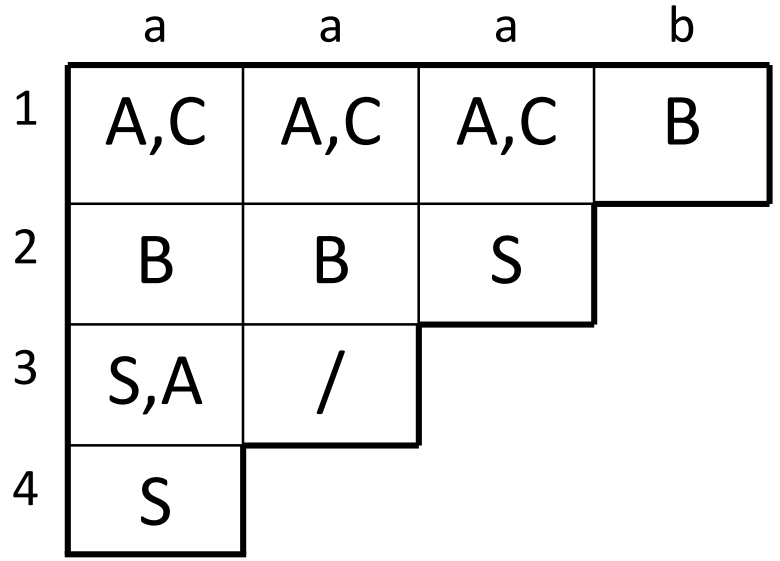
- Let $w = a_1 \ldots a_n$.
- We construct an n-by-n triangular array of sets of variables.
- X_{ij} = {variables A | A =>* $a_i \ldots a_j$}.
- Induction on j–i+1.
  - The length of the derived string.
- Finally, ask if S is in X_{1n}.

CYK Algorithm – (2)

- Basis: $X_{ii} = \{A \mid A \rightarrow a_i \text{ is a production}\}$.
- Induction: $X_{ij} = \{A \mid$ there is a production $A \rightarrow BC$ and an integer k, with $i \leq k < j$, such that B is in X_{ik} and C is in $X_{k+1,j}$.

CYK example

- $S \rightarrow A\ B$
  $A \rightarrow BC \mid a$
  $B \rightarrow CC \mid b$
  $C \rightarrow a$
- ? $S \rightarrow aaab$



CYK exercises

- $S \rightarrow A\ C \mid B\ D \mid A\ E$
  $C \rightarrow B\ B$
  $D \rightarrow A\ A$
  $E \rightarrow B\ A \mid A\ B$
  $A \rightarrow a \mid A\ E \mid E\ A \mid B\ D$
  $B \rightarrow b \mid B\ E \mid E\ B \mid A\ C$
- ? $S \rightarrow baabba$

- $S \rightarrow P\ N \mid other$
  $P \rightarrow I\ E$
  $I \rightarrow if$
  $E \rightarrow expression$
  $N \rightarrow T\ S$
  $T \rightarrow then$
- is the sentence correct
  $S \rightarrow$ if expression then if expression then other

Earley parser

- Jay Earley, 1968
- read symbol by symbol and tries all possible allowed production
- the parser stores a list of partially completed grammar rules for each position
- as words are taken in from the left-most position, the parser first determines what new grammar rules could start with a word of that type, and those rules are put into the list.
- the parser determines if a partially-completed rule that is already in the list needs a word of that type in that position to complete itself further.
- If so, the rule is taken out and replaced with the more complete version of itself.
- When a word fully completes a rule, it is taken out, replaced with the non-terminal corresponding to that rule, and the rule completion process is repeated, using the non-terminal to complete rules instead of the word.
- When complete, any sentence non-terminals that encompass the entire string are treated as valid parses for the sentence.

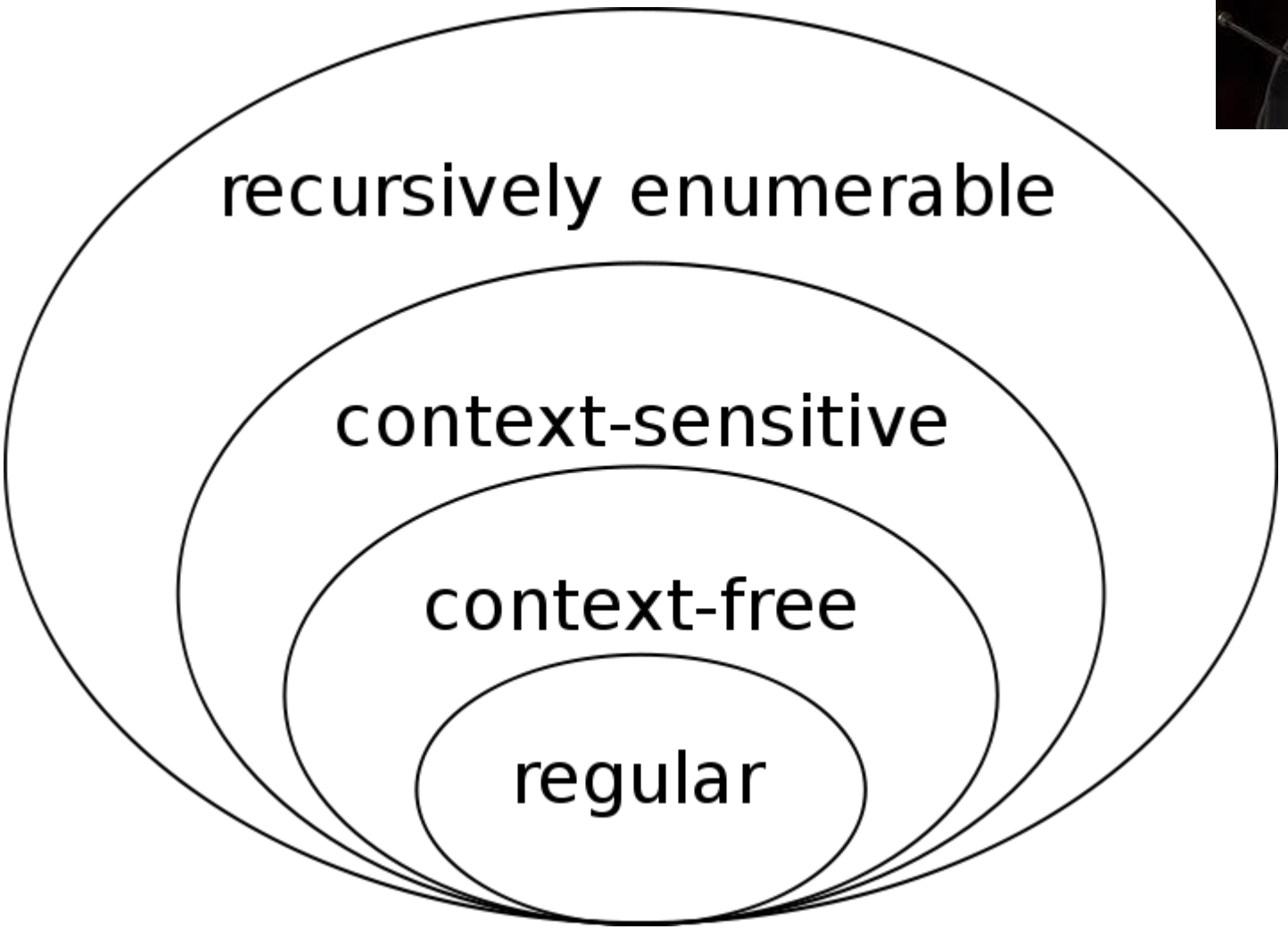
Earley parser - complexity

- $O(n^3)$ where *n* is the length of the parsed string,
- $O(n^2)$ for unambiguous grammars
- O(n) for almost all LR(k) grammars.
- performs particularly well when the rules are written left-recursively.

Practical use of grammars

- gnu program bison and yacc
- based on CFG generates a recognizer code in C, C++, or java

Chomsky hierarchy



Order 3

- Order 3 grammars are regular languages
- Grammars of the form

$S \rightarrow aA$
$S \rightarrow a$

Order 2

- CFGs
- Form $A \rightarrow \alpha$
- α is a string of terminals and nonterminals
- programming languages

Order 1

- Context dependent grammars CDG
- Form $\alpha A\beta \rightarrow \alpha\gamma\beta$
- A is a variable, α, β, and γ are strings of terminals and nonterminals
- α and β can be empty, γ has to be non-empty
- natural languages

Order 0

- Unbounded (Turing) grammars and Turing languages, i.e., languages recognizable by Turing machines
- Form $\alpha \rightarrow \beta$

- There are languages unrecognizable with Turing machines – diagonal proof